

Filtering Techniques to Improve Trace-Cache Efficiency

Roni Rosner, Avi Mendelson and Ronny Ronen

Microprocessor Research Lab, Israel Design Center

Intel Corporation

{ roni.rosner , avi.mendelson , ronny.ronen } @ intel.com

Abstract

The trace cache is becoming an important building block of modern, wide-issue, processors. So far, trace cache related research has been focused on increasing fetch bandwidth. Trace-caches have been shown to effectively increase the number of “useful” instructions that can be fetched into the machine, thus enabling more instructions to be executed each cycle. However, trace cache has another important benefit that got less attention in recent research: especially for variable length ISA, such as Intel’s IA-32 architecture (X86), reducing instruction decoding power is particularly attractive. Keeping the instruction traces in decoded format, implies the decoding power is only paid upon the build of a trace, thus reducing the overall power consumption of the system.

This paper has three main contributions: it indicates that trace cache optimizations directed to reducing power consumption are do not necessarily coincide with optimizations directed to increasing fetch bandwidth; it extends our understanding on how well the trace cache utilizes its resources and introduces a new trace-cache organization based on filtering techniques. The knowledge obtained from the analysis of the traces’ behavioral patterns motivates the use of filtering techniques. The new trace-cache organization increases the effective instruction-fetch bandwidth in conjunction with reducing the power consumption of the trace-cache system.

We observe that (1) the majority of traces that are inserted into the trace-cache are rarely used again before being replaced; (2) the majority of the instructions delivered for execution originate from the fewer traces that are heavily and repeatedly used; and that (3) techniques that aim to improve instruction-fetch bandwidth may increase the number of traces built during program execution. Based on these observations, we propose splitting the trace cache into two components: the filter trace-cache (FTC) and the main trace-cache (MTC). Traces are first inserted into the FTC that is used

to filter out the infrequently used traces; traces that prove “useful” are later moved into the MTC itself. The FTC/MTC organization exhibits an important benefit: it decreases the number of traces built, thus reducing power consumption while improving overall performance. For medium-size applications, the FTC/MTC pair reduces the number of trace builds by 16% in average.

An extension of the filtering concept involves adding a second level (L2) trace-cache that stores less frequent traces that are replaced in the FTC or the MTC. The extra level of caching allows for order-of-magnitude reduction in the number of trace builds. Second level trace cache proves particularly useful for applications with large instruction footprints.

1. Introduction

Modern high-end processors require high throughput front-ends in order to cope with increasing performance demand. Fetching a large number of useful instructions (instructions residing on the correct path) per cycle is a non-trivial task since, on average, every fifth instruction is a branch. In the current instruction-cache organization, fetching a large number of useful instructions per cycle requires accurate prediction of multiple branch targets and the assembly of instructions from multiple non-contiguous cache blocks.

One way of increasing the effectiveness of conventional fetch is reducing the number of fetch discontinuities along the execution path of the program. Loop unrolling [7], code motion [32], branch alignment [11] and software trace caches [20] are all examples of compiler techniques that reduce the number of taken branches and, subsequently, reduce the amount of instruction stream discontinuities in programs executions. Special ISA’s (Instruction Set Architectures) [15][28] have also been proposed to better express such compiler optimizations to the hardware. However, the illusion of continuous fetch can also be constructed at run-time. The **trace-cache** [18][22][23][24][25] is an instruction memory component that stores instructions in the order

they are executed rather than in their static order as defined by the program executable.

Taking advantage of the fact that programs execute the same instruction paths repeatedly, the trace cache stores frequently executed non-contiguous instruction blocks as straightened out runs called *traces*. When these instructions subsequently need to be fetched, the complex operation of multiple branch prediction can be done in parallel to the fetch, assuming speculatively the repeated execution of a previous trace [10]. In addition, multiple block assembly can be skipped because the results of these actions are already implicit in the trace itself.

The performance benefit of the trace cache extends beyond fetch convenience, however. Consider the processor non-specific stages, i.e. stages in which processing depends on the static instruction only and not on its dynamic instance, like instruction decode. The presence of an interim form of the instructions, as exist in a trace cache, facilitates decoupling of these stages from the common-case processing path, by storing pre-processed traces that already reflect the operation of these stages. Essentially, these pipeline stages can be moved in front of the trace cache so that their latency is only observed on a trace-cache miss. This feature is especially useful when the pipeline stage in question is long, complex and power-hungry, like the decoding stage for Intel's IA32 architecture [13][14][30].

The functionality of the trace-cache can be divided into *trace-building* that examines the dynamic instruction stream, selects and collates common instruction sequences, and *trace-bookkeeping*, which attempts to maintain the trace working set in the trace cache and avoid unnecessary re-builds. Most recent research on trace caches has focused on performance aspects – improvements to the trace-build process [8][16][26][27], the format “convenience” of the output instruction stream [5][6][8][17] or on finding new structures for the trace [1][5][12]. A performance-perspective study of trace cache limitations can be found in [19].

However, trace caches may play another important role in the system by reducing the power consumption of the processor. Power and energy consumption are becoming a major concern of processors at all performance levels, not just at the low end. Modern implementations of CISC architectures pose a particular challenge on the design of the processor's front-end (instruction fetch and decode) whose power consumption may get as high as 28% of the overall processor power [14]. Our study focuses on aspects of trace caches that may contribute to power and energy saving by the processor.

This paper examines trace-caches from several viewpoints. We start by studying the basic behavioral patterns of the traces within the trace-cache. We measure several fundamental characteristics of the traces

including: the frequency at which traces are used, their lifetime within the cache, their space utilization and how long it takes before a replaced trace is re-built. We also examine how the physical characteristics of the cache affect these parameters and use a near-perfect algorithm to examine the impact of the replacement mechanism. These observations help us converging on the idea that traces should be filtered in order to improve the trace-cache utilization.

Based on the above observations, we propose a new trace cache organization. We split the trace cache (TC) into two sections: the *Filter Trace Cache* (FTC) that filters out the infrequent traces, and the *Main Trace Cache* (MTC) that keeps the frequent ones.

Filtering techniques have already been proposed to improve the performance of cache-based systems. In [29] it is proposed to use a filter cache in order to improve the performance of data cache, and [21] suggests to “filter” traces that do not contain taken branches since they do not contribute to the instruction-fetch bandwidth. In this paper, we propose filtering techniques based on the *usage frequency* of traces aimed at reducing the traces build-rate in the system. The new trace cache organization is examined for its impact on the instruction-fetch bandwidth and on the *power saving*. We show that the optimization points for power saving is different than that for fetch bandwidth.

We observe that for medium size applications, the FTC/MTC organization reduces the build-rate by 16% in average, achieving increased performance and reduced power consumption. Several applications, typically those with large instruction footprints, exhibit smaller improvement. For these applications we propose using a second-level trace-cache (L2) in order to store evicted traces. We show that for applications with large instruction footprints, this trace-cache organization effectively reduces the number of builds by a factor of three, compared to the basic FTC/MTC organization.

The rest of the paper is organized as follows: Section 2 details the trace-cache model and the simulation environment. Section 3 describes various trace characteristics, focusing mainly on the characteristics of a single example - the gcc application. Section 4 presents the filtering mechanism and extends the cache organization with a second-level trace-cache. Finally, Section 5 concludes the paper.

2. Trace-Cache Model

We assume an abstract model of a machine in which instruction processing occurs in three major sub-systems:

- The *trace-building sub-system* fetches instructions from conventional instruction memory, decodes them and groups decoded instructions into *traces*.

Traces are stored in a trace-memory, which is typically structured as a trace-cache. In our model, every executed instruction comes from a trace that has been built and stored in the trace cache memory.

- The *trace-management sub-system* fetches whole traces from trace-memory, creates a continuous stream of their included instructions, and feeds that stream to the execution sub-system.
- The *execution sub-system* consumes the instruction stream.

The trace-management sub-system plays critical roles in both the performance and power domains. On one hand, it is expected to provide high throughput and a correct stream of instructions to the execution sub-system. On the other hand, it is expected to limit the use of the trace-building subsystem, which typically imposes long delays and consumes power at a high rate. One way to achieve both goals is to keep the “right” traces in trace memory so that the required traces are available for fast delivery as often as possible, and expensive re-builds and re-decodes are avoided as much as possible.

2.1. Limitations of the framework

The presented study was performed with a certain abstraction level in mind. It was designed to evaluate several new concepts at a rather high abstraction level. We present our simplifying assumptions, discuss their implications, and provide some justifications.

Since the penalty for failing to supply a trace when one required is very high, the mechanism for predicting the next trace to be fetched from (and into) the trace memory is critical. In this study, we assume perfect (*oracle*) prediction that guarantees we always fetch in advance and supply the “right” next trace for execution. The mechanisms that we evaluate in this study are quite orthogonal to the problem of trace prediction so it is safe to assume that the trade-offs we establish remain valid in the presence of realistic trace prediction. This assumption is supported by the limit studies reported in [19].

It should be noted that for simplicity – in order to abstract away from lower-level details of micro-architecture implementation – current results and defined metrics refer to traces built out of IA32 instructions rather than micro-operations. This abstraction is supported by the fact that typical IA32 applications exhibit an average of about 1.3 micro operations per each IA32 instruction (cf. [12]). Our definitions and metrics will retain their usefulness when applied to the more accurate level of micro-operations. We also expect that the trends and trade-offs presented hereafter will retain their validity in the more specific context.

2.2. Trace definitions

For brevity of description we lay down the following set of trace-related definitions. An **address** is a location in the application executable image at which a basic block begins. A trace may contain multiple basic blocks and multiple traces may begin at the same basic-block address. A trace is uniquely identified by a **tag** - a composite identifier that names all the conditional branches along a trace. However, for compactness of representation a tag is not a list of the instruction addresses in a trace. Rather, it is the address of the first basic-block and a list of n descriptors that specify the branch transition from one basic-block to another for a maximum of n branches in the trace. Transition descriptors are two bits long to specify the three block transition possibilities: transition via a taken branch, transition via a not-taken branch and the end of the trace.

A **trace frame** is the space in a trace-cache required to store a maximal-size trace. A **build** is the operation that consumes instructions fetched from the instruction cache, decodes them, and places the decoded instructions into a trace frame. An **access** is an instance at which a trace that is present in the trace-memory is extracted for execution. We make the simplifying assumption that the size of a trace (and that of a trace frame) is directly proportional to the number of instructions it encapsulates.

2.3. Trace building

Trace building is an iterative process in which instructions are fetched, decoded, and added to the trace. A trace always starts at an address of a branch target, and ends when a termination condition is met. The **termination criterion** is the factor that most directly controls the quality and utility of created traces. In general, traces are composed of an integral number of basic blocks. We do not allow breaking basic blocks or splitting them between different traces, except very exceptional cases such as a single very long basic-block.

In the current study we apply a composite criterion consisting of several fixed conditions and a few configurable parameters. The fixed conditions are:

- **Maximal number of branches:** A trace cannot contain more than 8 conditional branches. Each conditional branch along the trace contributes to the trace tag, and the tag is limited to 8 branches. The number of branches is roughly the number of basic blocks, if we ignore undetected branch-targets inside the trace. (Actually, the average number of basic blocks per trace is much smaller. Our detailed observations on such type of trace characteristics will be published elsewhere.)
- **Call:** Any procedure-call instruction terminates a trace.

- **Ret:** Any return from procedure instruction terminates a trace.
- **Interrupt:** Any asynchronous event that interrupts the normal execution of the program (e.g. a page fault) terminates a trace.

The configurable conditions are:

- **Trace capacity:** A trace cannot contain more than a predefined, fixed number of IA32 instructions. The considered configurations allow a maximum of 16 (these configurations are denoted by **I16**) or 32 (denoted by **I32**) instructions.
- **Backwards-Branch Termination Test:** here we have three options
 - o **Basic (B):** A trace terminates on any backward branch; this choice creates traces that are single loop iterations.
 - o **Non-Backward (nB):** A trace is not terminated on backward branches; this choice allows the creation of longer traces that may contain more instructions than those contained in a single iteration of a short loop.
 - o **Loop Unrolling (LU):** the build mechanism employs simple loop unrolling heuristics and its oracle knowledge of simple-to-detect loops in order to optimize the way certain loops are unrolled into traces.

The main difference between the last two mechanisms is that the **LU** mechanisms tries to fit a maximal number of whole loop-iterations into a single trace, while the simpler **nB** mechanism, being ignorant of iterations, fits the maximal number of instructions into a trace.

2.4. Trace-Cache Organization

A trace cache consists of controls and data area. The control of each cache entry holds a trace identifier including the starting address and branch conditions. In addition, the control keeps statistics on trace usage. The data area is capable of storing one trace per entry. We consider trace caches that are 8-way set-associative, except for the cases where we study the impact higher-associativity. A set is managed using an LRU replacement policy. The mapping of a trace into its hosting set is a function of its complete tag.

For the purpose of limit studies, we also consider less practical organizations, such as 16-way set-associative and fully associative caches, and will also examine near-perfect replacement policy.

In this study we consider systems composed of one or more trace caches. For example, when a filter cache is used, the evicted traces are directed to a cache-management logic together with their usage counters for further processing, as described in subsequent sections. An important functionality of the cache-management

logic is that of filtering traces according to a static or dynamic criterion. We call such a criterion a *filter*. In a later section, we shall define several types of filters and investigate their behavior.

We define a trace-cache to be *executing* if it delivers instruction from traces directly to the execution subsystem. Another type of trace cache we investigate is a *storage* trace caches, which is used as secondary trace storage (e.g., second level trace cache). The processor cannot fetch instructions from the stored traces directly - traces need to be transferred to an executing trace cache before their instructions can be issued for execution.

2.5. Data Collection and Evaluation Criteria

In the following few sections, we investigate several organizations for the trace cache. Throughout these studies, we assume a maximum data area of 128KB for all the executing caches in the system. Alternative organization techniques are compared according to the effectiveness of their exploitation of a given area. The area required for a cache equals:

$$(\text{\#of sets}) * (\text{\#of ways}) * \text{MAX_INST_IN_TRACE} * \text{INST_SIZE}$$

Here **MAX_INST_IN_TRACE** equals either 16 or 32 (I16 or I32 configurations, respectively), and **INST_SIZE** has a fixed size of 8 bytes. Thus, for instance, in an I16 configuration with a single 8-way associative cache, the number of sets is 128. In an I32 configuration with two equal-sized execution caches, the number of sets in each cache is 32. Note that storing **decoded** instructions implies an area increase, which is approximated by using the longer **INST_SIZE** (8 bytes).

Later on, we extend the study by considering the addition of second-level, or storage, caches. Then, we consider several alternatives of allocating area for the caches in the system.

Our evaluation relies on counting three kinds of events: *builds*, *accesses* and *transfers* (transfer is the event in which a trace is copied from one cache to another). Our main performance and power related metrics are:

- The trace-cache *instruction-fetch bandwidth*, or just *fetch-bandwidth* in short, computed by dividing the number of instructions by the number of trace cache accesses.
- The *build-rate*, defined as the average number of trace builds per k-instructions.
- The *transfer-rate*, defined as the average number of inter-cache moves per k-instructions.

Higher fetch-bandwidth correlates with higher performance. Build-rate affects both performance and power. Building a trace involves fetching and decoding of raw instructions - it has higher latency and lower fetch-

bandwidth than fetching decoded instructions from the trace cache and it consumes a lot of power. Consequently, higher build rate implies lower performance and higher power consumptions. Transfer-rate also affects both performance and power - but to a much lesser extent - transferring an already built trace is basically a simple copy and it is a much faster and less power consuming operation than building a trace. As we will show, the techniques described in this paper affect mainly the build-rate and the transfer-rate. To gain performance and reduce power, we strive to mainly decrease the build-rate. Decreasing the transfer-rate does help, but it is much less crucial.

2.6. Simulation Environment

Our simulation environment is composed of four types of components, as depicted in Figure 2-1:

- **Trace Builder:** Generates traces out of a dynamic execution stream, according to the trace-build options.
- **Cache Manager:** Controls the transfer of traces between the different caches, according to the different filtering policies and cache organizations.
- **Cache:** Manages the storage and supply of traces according to its configuration (#of sets, associativity, LRU policy, etc). Our environment may contain one or more caches.
- **Statistics Manager:** Gathers data on the interesting events in the system and outputs the requested histograms.

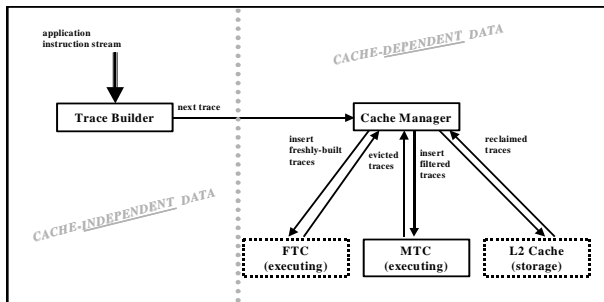


Figure 2-1 Simulation Environment. The cache-independent component is responsible for trace construction out of the instruction stream, and is affected by build configuration parameters. The cache-dependent components simulate the effects of the different cache logic and storage in the system. They are affected by the different cache organization, size, associativity, replacement and filtering parameters studied.

Our set of benchmarks is composed of ten application traces consisting of 30 million instructions each. The instruction-traces consist of representative portions of the applications sampled within their execution. The applications belong to several benchmarks:

- SPEC-CPU INT 2000: gcc, vortex, crafty.
- MMmark99-Win98: video.

- SYSmark98-NT4: mpeg, photoshop, ppt.
- SYSmark00-Win2k: excel, word.
- Speech recognition.

3. A Study of Trace Behavior

In this section we characterize different trace behaviors within the trace cache. In order to guarantee the generality of our results, we do not limit the study to any specific implementation (such as the trace cache of the Intel Pentium® 4 Processor [31][30] or the Sparc64 V [3]). For example, our trace building mechanism allows traces with the same starting address but differing in internal branches outcome to coexist in the trace cache. This technique makes our results less sensitive to the quality of a certain branch predictor and to the effects of miss-speculated paths.

Results in the paper are presented in two manners. When a broader understanding is required, we present the results from the complete benchmark suite. For other experiments, mainly when a deeper analysis of the behavior is required, we demonstrate the results using a single program: *gcc*.

3.1. Increasing Instruction-Fetch Bandwidth vs. Reducing Build-Rate

Optimizing trace cache for performance is different than optimizing it for power reduction. Performance optimizations are mainly concerned with increasing the number of “useful” instructions supplied to the core, while power-aware optimizations are directed at reducing the frequency at which new traces need to be built. This section focuses on the tradeoffs between these two optimization points. Recall that *fetch-bandwidth* was defined as the average number of instruction fetched from a trace throughout the execution of an application.

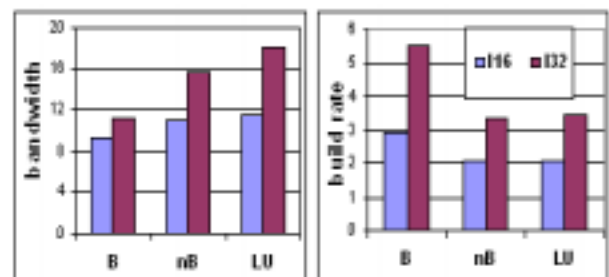


Figure 3-1: Left: *fetch-bandwidth* measured in instructions per trace access, comparing different trace sizes and different trace-building mechanisms. Right: *build-rate* measured in trace-builds per K instructions (gcc).

Figure 3-1 shows the fetch bandwidth (correlated with performance) and the build-rate (correlated with power) of a system, when running the GCC benchmark and

comparing two different optimizations: (1) increasing the trace size (while keeping the capacity of the caches fix) and (2) using more sophisticated trace-termination conditions.

As expected, both trace size and the degree of loop unrolling increase the number of instructions produced from a trace, explaining the impact on fetch bandwidth. Note that the benefit of unrolling (**LU** vs. **B**) is significantly higher for larger traces (**I32** vs. **I16**).

The effect of backward-branch policy and trace size on the build-rate is less intuitive. The number of builds increase with trace size due to an effective decrease in trace cache capacity. Recall, traces can overlap, and longer traces increase the likelihood of overlap. At the same time, longer traces reduce the number of traces and the flexibility in managing the increased overlap. The end result is that the cache contains more redundancy, reducing its effective capacity and requiring more builds. The interaction with loop unrolling is negligible for short traces, but significant for longer ones; excluding loop unrolling dramatically increases the number of builds. The reason for this is simple as well. Forcing traces to terminate at every backward branch (**B**) creates many short traces and drastically reduces the utilization of individual traces. Again, this effect is magnified for the longer traces.

3.2. Lifetime characterization of traces

Another important metric is the lifetime of a trace in the cache. Specifically, we are interested in the duration at which a trace **resides** in the cache and the **renew** time, which is the time from its replacement until the next time it is brought into the cache. The residence period is further partitioned, as follows. The trace **live** time is measured from its entrance to the cache until the last access to this particular occurrence, and the **decay** time is measured as the time from the last access to the trace until it is replaced. The decay time represents the duration at which the trace was **dead** in the cache. Figure 3-2 displays the trace lifetime for different configurations.

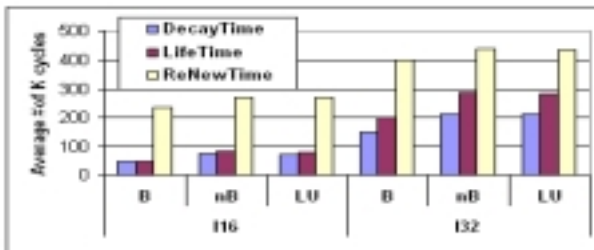


Figure 3-2: Breakdown of *trace lifetime* within the trace cache system (gcc). *LifeTime* is the useful period between first and last access to the trace, *Decay-Time* is the subsequent time until replacement, and *ReNew-Time* is the time between eviction and next build. Time is given in thousands of cycles.

We notice that the time the trace spent in the trace is almost equally divided between its live time and its decay time. This indicates that a different replacement strategy could potentially improve the utilization of the trace cache system. Another important observation is that the renew time; i.e., the time a trace stays outside the cache before being next consumed, is relatively long. This observation will support our proposed techniques, as described in the next section.

A closer look at Figure 3-2 reveals that: (1) the lifetime of larger traces is significantly longer than the lifetime of the shorter traces, (2) unrolling enlarges the lifetime of traces, and (3) nB and LU unrolling techniques have similar effect on the residency duration for both trace lengths.

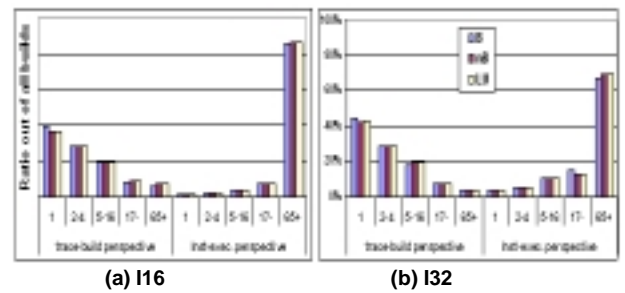


Figure 3-3: Distribution of traces according to number of accesses per build (gcc). The *trace-build perspective* shows the percentage of all builds at each range of accesses. The *instruction-execution perspective* shows the percentage of issued instructions at each range of trace-accesses.

A final set of data that motivate our proposed organization is the contribution to total accesses and total instructions fetched broken down by trace-builds, where trace-builds are grouped by the number of accesses before replacement. Figure 3-3 shows two charts, one for traces of length 16, the other for traces of length 32. Each chart has two parts; the left is a breakdown of the percentage of total accesses. The group of bars on the left represents the fraction of total accesses contributed to traces that were accessed only one time before they were evicted. The second group is the fraction of total accesses contributed by traces that were accessed twice, three, or four times before they were evicted and so on. The second part of the chart breaks down the total number of instructions fetched in the same way. The trend of these graphs is very clear – although 40% of the traces are referenced just once before eviction (and around 70% are accessed at most four times), the vast majority of instructions come from those few traces that are executed many times. We observed similar behavior in all the programs we examined.

3.3. Limit study

In this section we evaluate the impact of close to optimal replacements algorithms (cf. [1]), and the impact

of the degree of associativity of the cache on the utilization of traces within the trace cache. This behavior may suggest directions for potential improvement to trace cache utilization.

We define perfect cache replacement as the policy of replacing the trace whose next access is the latest, among the traces in the set. Near-perfect replacement is the policy of replacing the trace whose next access, within a given look-ahead window into the future, is the latest. In the event that several traces have their next access beyond the look-ahead window, we employ the standard LRU mechanism to select among these traces.

In this study we consider look-ahead windows of 1k-traces, 100k-traces and 1M-traces. As can be seen from the data below, the phenomena we investigate converge at look-ahead of 100K or 1M, so there is no practical reason to go beyond these limits or to consider absolute perfect replacement.

To examine the impact of increased associativity, we consider two enhanced cases: double associativity, i.e. using 16 ways instead of 8, and full associativity. Note that we are using a fixed die-size, meaning that twice the associativity implies half the number of sets.

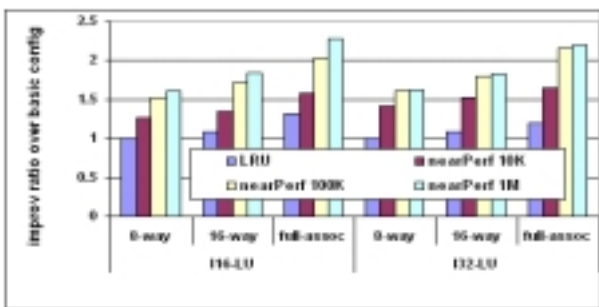


Figure 3-4: Limit study: impact of replacement policies and associativity on build-rate reduction (geometric mean of ten programs)

In order to expose the full picture of limits, we present the results of cross-using near-perfect and improved associativity. Figure 3-4 presents the results of these experiments in terms of build-rate, while Figure 3-5 presents the impact on the average trace lifetime.

We observe that although both increased associativity and perfection of the replacement policy improve the build-rate, the impact of the replacement policy is much stronger. Unfortunately, perfect replacement algorithms are not implementable. However, these results indicate that techniques that may extend the average time a ‘useful’ trace resides within a cache, may improve the overall performance of the system.

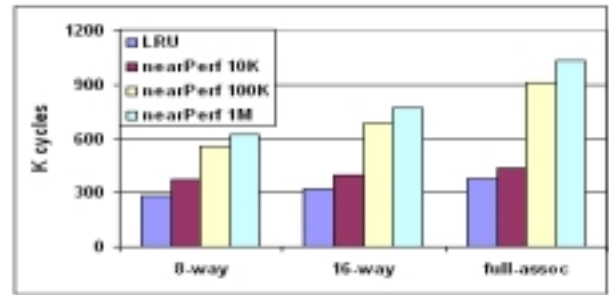


Figure 3-5: Limit case: impact of replacement and associativity on average trace lifetime (gcc l16 LU), measured in cycles. Notice that the significance of perfect replacement policy is revealed with a look-ahead window of 100k traces.

3.4. Conclusions and Observations

Our findings in this section can be summarized by the following observations:

- For a given build policy, optimizing the trace cache for instruction-fetch bandwidth increases the number of builds at run time. That is, performance oriented optimizations (higher bandwidth) may result in higher power consumptions (more builds).
- The majority of traces are used only few times before being evicted.
- The majority of instructions come from traces that have been used many times before being replaced from the trace cache.
- In average, a relatively long time passes before an evicted trace is needed again.

A new trace-cache organization that exploits these observations in order to improve trace-cache effectiveness is proposed in the next section.

4. Filtering

4.1. Concept

The last set of observations made in section 3, lead us to the conclusion that a filtering mechanism is needed in order to separate traces that are heavily used from those that do not show locality (in time) of reference. To implement this filtering, we separate the trace cache into two parts. The Filter Trace Cache (FTC) is used for filtering; the Main Trace Cache (MTC) stores traces that have been selected by the filtering process. This organization is shown in Figure 4-1. Both the FTC and MTC are 8-way associative, and each one of them has half the number of sets of the original, MTC-only configuration.

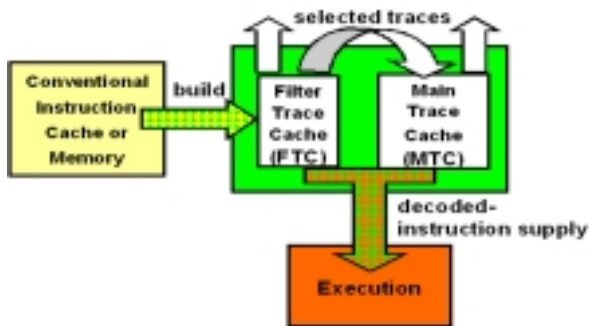


Figure 4-1: Basic filter system, composed of FTC and MTC. A filtering mechanism selects the ‘hot’ traces among those replaced in the FTC to be inserted into the MTC. All other replaced traces (from both FTC and MTC) are evicted from the cache memory system. This implies mutual exclusion between the traces residing in the FTC and MTC.

Instructions are fetched from the instruction cache, decoded, and built into traces. Built traces are entered into the FTC. Traces that are evicted from the FTC are either discarded or moved to the MTC for longer term storage. The decision to discard or to promote a trace is made based on a filter that implements some heuristic. A naïve filter may be based directly on the data shown in Figure 3-3. Namely, if a trace has been accessed more than a small constant (e.g., twice) prior to its FTC eviction, we assume that it is a useful trace, one we will continue to access again and again, therefore it gets promoted. Otherwise, we discard the trace on the grounds that it is likely to be one of the 60% of traces that once promoted will just be evicted from the MTC before being used again.

In the next section we consider several such filters, and present comparative results indicating that there is room for improvement beyond the naïve filter presented above.

4.2. Alternative Filters

In this section we present the impact of using different filters in the FTC/MTC trace-cache organization. The filter mechanism decides, upon the replacement of a trace from the FTC, whether to insert it into the MTC or to discard it. This mechanism can be either *static* or *dynamic*. A static filter simply compares the number of accesses of a trace to a fixed constant, and discards it if the access-count is smaller than this threshold. We denote by **C2**, **C3**, **C4** the filter whose assigned constant is 2, 3 and 4, respectively. **C1** is a filter that allows all traces to pass, assuming all traces in the cache have been accessed at least once (this assumption would no longer be true had we used any trace prefetching mechanism). Note that a filter system with **C1** is somewhat different than an MTC-only system: a trace transferred to the MTC is no longer subject to the LRU policy of the FTC.

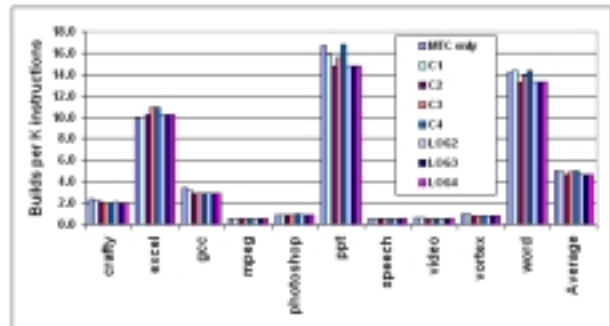


Figure 4-2: Build-rate for alternative filters (I32 LU), measured in trace-builds per K-instructions.

A dynamic filter represents a self-adjusting system that modifies its filtering criterion based on past feedback. We will present one simple type of dynamic filter which we call **LOG filter**. The LOG filter **LOG4** contains 3 registers: d1, d2 and d3. For each state of the registers we define a barrier b, defined as the largest j, such that $d_j > 0$. (if no register is positive, the barrier is $b=0$). The functionality is separated into two phases: filtering and update. Given the trace access-count N as input, the trace is allowed to pass if N is larger than the barrier. Following each filtering phase comes an update phase, at which N affects the next barrier. Each register d_j is incremented by $\log_2(N+2)$ if $N > j$, or decremented by this amount if $N < j$. Intuitively, the barrier represents an approximation to the average among all the past traces. The traces are weighted into this average by the logarithm of their the access-counts. Figure 4-2 presents the build rate for several static and dynamic filters on the full set of benchmarks, with traces of length I32.

The build behaviors observed in Figure 4-2 separate the benchmarks into three categories. In applications with small trace footprints, like *mpeg*, *speech* and *video*, the working set is so small that traces are essentially never evicted and only cold (first-time) builds are required. No optimization will reduce the number of these builds. Applications with medium trace footprints like *crafty*, *gcc*, *photoshop* and *Vortex*, have some eviction misses and present an opportunity for smarter trace-cache management provided by the FTC/MTC organization. We call them **medium-size** applications. The third category contains programs with large trace working sets like *excel*, *ppt* and *word*. The opportunity for smart management here is great but so is the likelihood that the total capacity of the FTC/MTC is simply not large enough to hold the entire working set, and that the optimal number of misses (builds) even for perfect trace replacement will be high. We call these **large footprint** applications.

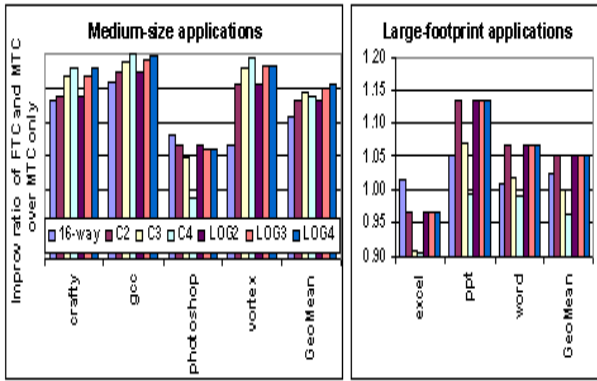


Figure 4-3: Benchmark summary of build-rate improvement in build-rate of alternative filtering techniques over the basic system of MTC-only (I32, LU). The left-hand graph displays the improvement ratio for the medium-size applications, while the right-hand graph shows the improvement for the large-footprint programs. Longer trace size show smaller improvement rate, though the relative quality of different filters remains more or less intact. For comparison, the leftmost bars (16-way) represent the impact of doubling the associativity of MTC from 8 to 16, while the other bars relate to filters on a FTC/MTC system, both 8-way associative.

Given the above classification, Figure 4-3 presents the final “score” of the filtering techniques, measured as the improvement ratio of each filter over the basic MTC configuration (higher ratio is better).

Firstly, we observe that the FTC/MTC organization is most beneficial for the medium-size applications, while for the large-footprint applications the improvements are more modest. When the trace working set is larger than the trace cache, many of the misses we see are true capacity misses, not replacement-policy-sensitive conflict misses. In order to improve trace-cache performance for these applications, in the next section we propose to add a second-level trace-cache that stores traces evicted from the FTC/MTC.

Secondly, we observe that the FTC/MTC partition of the system has a greater impact than mere increase in associativity. Except for photoshop and excel, the improvement gained by double-associativity is smaller than that obtained by most filters. In other words, two 8-way associative caches with LRU replacement and a filter in-between are a better cache organization, in terms of build rate (but also in terms of smaller LRU logic) than a single 16-way associative cache.

Focusing on the medium-size applications, we observe significant improvement to the build-rate by up to 20% over the conventional TC organization. Comparing the behavior of different filters, we note that the dynamic filter LOG4 (for I32LU) performs the best, scoring an average improvement of 16%. As these filters are global, they are much simpler to design and cheaper in terms of die-size than any replacement policy improvement that

has to be locally implemented for each of the cache sets. Compared to the results of the limit study presented in the previous section, we conclude that the demonstrated improvement is sub-optimal, leaving room for further improvement of the filtering techniques.

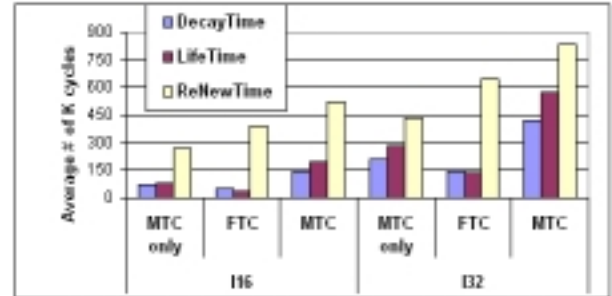


Figure 4-4: Trace lifetime breakdown: MTC only vs. FTC+MTC, measured in cycles. The filter clearly imposes a separation of traces into short-life ones (observed in the FTC) and longer-life traces (observed in the MTC).

4.3. Analysis of FTC Behavior

Additional indication on the special role played by the FTC can be obtained by observing the lifetime of traces in the FTC/MTC system. Figure 4-4 presents the lifetime breakdown of an MTC-only system vs. the components of an FTC/MTC system. The data presented in Figure 4-4 shows the significant improvement in the average period traces live in the FTC/MTC system; in particular, the improvement is exhibited by the MTC. This should be of no surprise since after proper filtering, the MTC is expected to contain more of the “hot” traces. This phenomenon is clearly observed in the distribution of execution and builds presented [4].

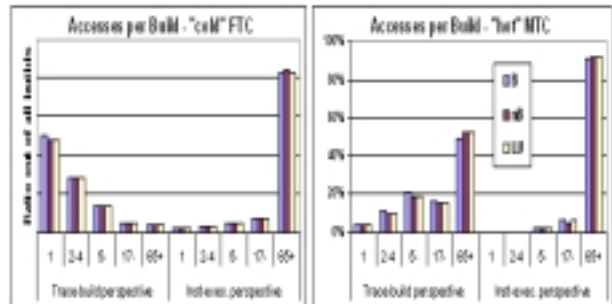


Figure 4-5: Accesses breakdown: FTC vs. MTC (gcc, I16). Compare to the MTC-only breakdown in Figure 3-3.

Another important aspect of the FTC/MTC organization is the fact that the improvement in build-rate is achieved in the cost of more data transfers between the FTC and the MTC. The data in Figure 4-6 shows the transfer rate for different filters, demonstrating that there are additional considerations to be evaluated when a choice of filter has to be made. For example, any IA-32

design would appreciate the reduction in the very costly builds, while a RISC design may prefer a different tradeoff.

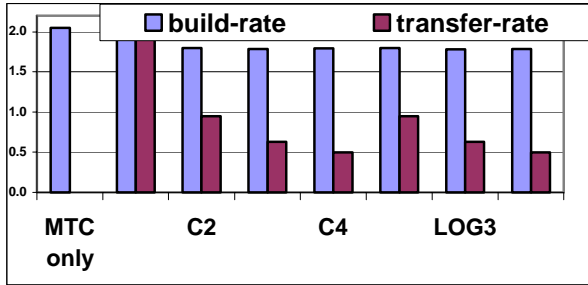


Figure 4-6: Build-rate vs. transfer-rate, measured in builds or transfers per K-instructions, as affected by alternative filters (gcc i16LU).

4.4. Hierarchy

As shown above, the trace filter mechanism reduces the number of trace builds by improving the trace replacement policy. Still, the number of trace builds, especially for the large-footprint applications remains high, mainly due to limited capacity. Reducing the number of trace builds for such capacity limited applications can be achieved by enlarging the trace storage. Unfortunately, bigger FTC/MTC pair involves higher latency and higher power, therefore, this section examines the benefit of adding a second level (L2) trace cache. The proposed L2 cache aims to enlarge the overall space devoted to keeping decoded trace caches. The L2 cache is assumed to be slower than the L1 trace cache, but also to consume significantly less power per byte. Using several levels of trace caches enables us to trade between area, performance and power consumption. We show that L2 trace cache significantly reduces the number of trace builds relative to the FTC/MTC only configuration. In particular, applications with large footprint enjoy significant reduction in the number of absolute builds.

In terms of cache-size, we investigate the following configurations. The base line is the 128KB FTC+MTC system. We investigate the option of progressively extending it with larger L2 caches. Specifically, we add L2 of sizes that are multiples of the original 128KB FTC+MTC, and denoted by 1*L2 (for the 128KB L2, and total of 256KB trace-cache memory) and 3*L2 (for the 38 KB L2, and total of 512KB trace-cache memory). All the L2 configurations are 8-way associative.

Again, we focus our measurements at the impact on build-rate. We assume that the operation of trace-build from conventional memory is far more expensive than bookkeeping operations such as the transfer of traces from one level in the hierarchy to another, both in terms of power consumption and in terms of added delays. We therefore argue that the power and performance gain

related to reduction in the number of builds due to the introduction of an L2 trace-cache are significantly higher than the loss due the extra bookkeeping overhead measured by a much larger number of transfers.

4.4.1. Results

The results of the L2 trace cache experiment are presented in Figure 4-7. We observe that the L2 trace cache significantly improves the build rate, up to 8 times reduction with the largest L2 considered. This trend is consistent for all applications. However, the absolute reduction in build rate is far bigger for the large-footprint applications. In view of the previous results on filtering, the L2 trace cache enables these applications to overcome the inherent capacity limitations of the ordinary trace-cache system. Comparing Figure 4-7 to Figure 4-3 we can see that even the build-rate of excel application, which could not be improved by any of the suggested filters, gets finally reduced by the L2 trace cache.

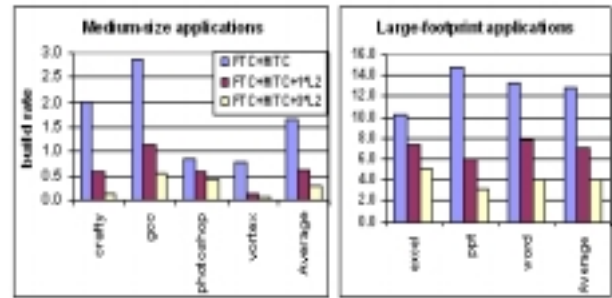


Figure 4-7: L2-size impact on build rate: The impact of L2 trace-cache is considered on systems with several cache-sizes, which are multiples of the basic FTC+MTC size (132 LU). Although the impact of progressively increasing L2 size is similar for both types of applications, the absolute reduction in build rate is significantly higher for the large-footprint applications (right).

The improvement ratios in build rate are presented in Figure 4-8. We obtain a reduction of over 3X for the large-footprint applications, and even larger reduction ratio for the medium-size ones.

A complete trade-off analysis should include the impact of the additionally introduced transfers. In a trace memory with L2 cache there is different price in terms of power consumption to transfers from FTC to MTC and to transfers from MTC to L2 (or vice versa). These parameters are design related and are not available to us at this point. Future work should extend the current study with a detailed analysis of the trade offs between power consumption, cost of extra area and the performance impact of adding L2 trace cache while reducing the size of the L1 trace cache.

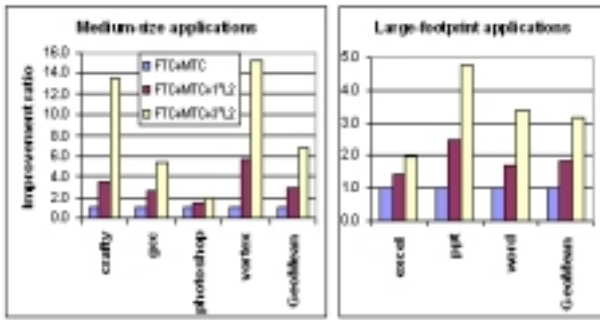


Figure 4-8: Reduction in *Build-rate*: the graphs show the *Improvement ratio* of a system with L2 over the basic FTC+MTC configuration, compared for various L2 sizes (I32 LU).

5. Conclusions and Future Work

This paper has three main contributions: it shows that optimizing traces for reduced power is different than optimizing them for high fetch bandwidth; it provides in-depth information on how traces are utilized within the trace cache, and it presents new microarchitectural ideas for improving the power consumption of trace-cache based systems. In the first part of the paper we analyze the *behavioral patterns* of traces within the trace-cache. The analysis indicates that:

- Different techniques are needed for optimizing the trace cache for power reduction (minimizing the traces build-rate) and for performance (maximizing the instruction-fetch bandwidth). A trace cache that is optimized for high instruction-fetch bandwidth requires complicated and power-hungry front-ends in order to cope with increasing performance demand. Such trace-caches tend to enlarge the size of the trace-line and use sophisticated loop unrolling mechanisms. Unfortunately, these techniques may result in increased build-rate and lower utilization of the trace-cache. Consequently, the power consumption of the front-end may significantly increase. Note that a large number of builds may harm the performance as well, unless expensive hardware (in term of design cost and power dissipation) is being used.
- The majority of the traces are rarely used. Many traces are built only once.
- The majority of the executed instructions come from the more frequently used traces.
- Good replacement mechanism has a significant effect on reducing the build-rate.

These behavioral patterns indicate that filtering techniques can increase the utilization of the trace cache by preventing the infrequently used traces from polluting the trace-cache. Another observation is that filtering also

enlarges the lifetime of the traces, thus contributing to the overall performance.

We propose a new trace cache organization based on the partition of the trace cache into FTC and MTC. As measured on medium size applications, the FTC/MTC organization improves the build-rate by 16% in average, achieving increased performance and reduced power consumption. We also evaluated the impact of adding second-level trace-cache (L2) that stores traces evicted from the FTC and MTC. We found that for applications with large instruction footprints, this trace-cache organization effectively reduces the number of builds by a factor of 3 compared to the basic FTC/MTC organization.

In view of the encouraging results of the filtering techniques presented in this paper, we believe that other filtering algorithms should be investigated. One possible direction is to change the hardware/software interface and use software directed “hot path” selection techniques similarly to the proposals in [4]. Additional micro-architectural aspects of the memory system recommended for future research include the impact of trace selection and branch prediction on the overall performance and energy consumption of the processor. Studying different alternative organizations and parameters related to the introduction of a large L2 trace cache may also provide important insight into the system’s area, performance and power tradeoffs.

Acknowledgements

The authors would like to thank Amir Roth for his careful review of an earlier version of the paper and for his very useful comments and suggestions.

References

- [1] L.A. Belady, “A Study of Replacement Algorithms for a Virtual Storage Computer”, in *IBM Systems Journal*, vol. 5(2), pp. 78-101, 1966.
- [2] B. Black, B. Rychlik, and J. Shen, “The Block-based Trace Cache”, in *Proc. 26th Intern. Symp. on Computer Architecture*, May 1999.
- [3] D. Diefendorff, “HAL Makes Sparcs Fly”, in *Microprocessor Report*, vol. 13(15), Nov. 1999.
- [4] E. Duesterwald and V. Bala, “Software Profiling for Hot Path Prediction: Less is More”, in *Proc. of ASPIS-VI*, Nov. 2000.
- [5] D. Friendly, S. Patel and Y. Patt, “Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism”, in *Proc. 30th Intern. Symp. on Microarchitecture*, Dec. 1997.
- [6] D. Friendly, S. Patel and Y. Patt, “Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors”, in *Proc. 31st Intern. Symp. on Microarchitecture*, Nov. 1998.

- [7] J.C. Huang and T. Leng, "Generalized Loop-Unrolling: a Method for Program Speedup", in *Proc. IEEE Symp. on Application-Specific Systems and Software Engineering and Technology*, 1999.
- [8] Q. Jacobson, "High-Performance Frontends for Trace Processors", Ph.D. Thesis, Department of Electrical & Computer Engineering, Univ. of Wisconsin – Madison, Aug. 1999.
- [9] Q. Jacobson and J.E. Smith, "Trace Preconstruction", in *Proc. 27th Intern. Symp. on Computer Architecture*, June 2000.
- [10] Q. Jacobson, E. Rotenberg and J.E. Smith, "Path-Based Next Trace Prediction", in *Proc. 30th Intern. Symp. on Microarchitecture*, Dec. 1997.
- [11] Q. Jacobson and J.E. Smith, "Instruction Pre-Processing in Trace Processors", in *Proc. 5th Intern. Symp. on High Performance Computer Architecture*, Jan. 1999.
- [12] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, "eXtended Block Cache", in *Proc. 6th Intern. Symp. on High Performance Computer Architecture*, Jan. 2000.
- [13] K. Krewell, "Quicktake: Willamette Revealed", *Microprocessor Report*, Feb. 2000.
- [14] S. Manne, D. Grunwald and A. Klauser, "Pipeline gating: Speculation Control for Energy Reduction", in *Proc. 25th Intern. Symp. on Computer Architecture*, pp. 132-141, June 1998.
- [15] S.W. Melvin and Y.N. Patt, "Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines", in *Proc. Intern. Conf. on Supercomputing*, pp. 427-432, 1989.
- [16] S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism", Univ. of Michigan Technical Report CSE-TR-335-97, 1997.
- [17] S. Patel, M. Evers, and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing", in *Proc. 25th Intern. Symp. on Computer Architecture*, June 1998.
- [18] A. Peleg and U. Weiser. "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", U.S. Patent 5,381,533, Jan. 1995.
- [19] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of Instruction-Level Parallelism*, vol. 1, Oct. 1999.
- [20] A. Ramírez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache", in *Proc. Intern. Conf. on Supercomputing*, pp. 119 – 126, 1999.
- [21] A. Ramirez, J.L. Larriba-Pey and M. Valero, "Trace Cache Redundancy: Red and Blue Traces", in *Proc. 6th Intern. Symp. on High-Performance Computer Architecture*, pp. 325-333, 2000.
- [22] E. Rotenberg, "Trace Processors: Exploiting Hierarchy and Speculation", Ph.D. Thesis, Univ. of Wisconsin, 1999.
- [23] E. Rotenberg, S. Bennett and J.E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", in *Proc. 29th Intern. Symp. on Microarchitecture*, Dec. 1996.
- [24] E. Rotenberg, S. Bennett and J.E. Smith, "A trace cache microarchitecture and evaluation", in *IEEE Trans. on Computers*, 48(2), pp. 111–120, Feb. 1999.
- [25] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, "Trace Processors", in *Proc. 30th Intern. Symp. on Microarchitecture*, Dec. 1997.
- [26] E. Rotenberg, Q. Jacobson and J.E. Smith, "A Study of Control Independence in Superscalar Processors", In *Proc. 5th Intern. Symp. on High Performance Computer Architecture*, Jan. 1999.
- [27] E. Rotenberg and J.E. Smith, "Control Independence in Trace Processors", in *Proc. 32nd Intern. Symp. on Microarchitecture*, Nov. 1999.
- [28] M.S. Schlansker and B.R. Rau, "EPIC: Explicitly Parallel Instruction Computing", *Computer*, vol. 33(2), pp. 37-45, Feb. 2000.
- [29] J. Sahuquillo and A. Pont, "The Filter Cache: A Run-Time Cache Management Approach", in *Proc. 25th EUROMICRO Conference*, vol. 1, pp. 424 –431, 1999.
- [30] T. Pabst, "Intel's New Pentium 4 Processor" in <http://www2.tomshardware.com/cpu/00q4/001125> (Tom's Hardware Guide), Nov. 2000.
- [31] M. Upton, "The Intel Pentium® 4 Processor", in <http://www.intel.com/pentium4>, Oct. 2000.
- [32] C. Young, D.S. Johnson, M.D. Smith, and D.R. Karger; "Near-Optimal Intraprocedural Branch Alignment", in *Proc. 1997 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1997.